Decomposing God Classes at Siemens

Nicolas Anquetil, Anne Etien, Gaelle Andreo, Stéphane Ducasse

Decomposing God Classes at Siemens

Nicolas Anquetil*†, Anne Etien*†, Gaelle Andreo‡ and Stéphane Ducasse†*
*Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 - CRIStAL,
59650 Villeneuve d'Ascq, France
Email: firstName.lastName@univ-lille.fr
†RMod team
INRIA Lille Nord Europe,
Villeneuve d'Ascq, France
Email: firstName.lastName@inria.fr
‡Siemens AG
Digital Industry Division
Erlangen, Germany
Email: firstName.lastName@siemens.com

***Abstract*—A group of developers at Siemens Digital Industry Division approached our team to help them restructure a large legacy system. Several problems were identified, including the presence of God classes (big classes with thousands of lines of code and hundred of methods). They had tried different approaches considering the dependencies between the classes, but none were satisfactory. Through interaction during the last three years with a lead software architect of the project, we designed a software visualization tool and an accompanying process that allows her to propose a decomposition of a God Class in a matter of one or two hours even without prior knowledge of the class (although actually implementing the decomposition in the source code could take a week of work). In this paper, we present the process that was formalized to decompose God Classes and the tool that was designed. We give details on the system itself and some of the classes that were decomposed. The presented process and visualisations have been successfully used for the last three years on a real industrial system at Siemens.**

I. Introduction

Code smells, or anti-patterns, have long been identified as negative quality attributes (*e.g.,* [AKGA11], [DDVT06]) that hinder software evolution, and are sources of unnecessary costs. As a side note, it must be acknowledged that not all authors agree with that view. Olbrich *et al.,* [OCS10] state that "the presence of God and Brain Classes is not necessarily harmful; in fact, such classes may be an efficient way of organizing code."

Code smells have been the subject of many publications, to automatically identify them (*e.g.,* [AD15], [FTSC12], [LM06], [MCK15], [MSdMdSN14]), to remove them (*e.g.,* [BOG+14], [CLL13], [DDN02], [FTSC12], [LM06], [RR12], [SSL01], [SSS14]), or to analyse the quality gains their removal might bring (quality improvement [DDVT06], [FTSC12] or even Power Consumption improvement [PCP14]).

Our experience and anecdotal evidence tell us that in industry, numerous systems present a wide range of code smells. Successful industrial software systems are long lived and typically end up exhibiting various software quality problems including code smells. The reasons for this lack of quality are multiple:

- First some smells (*e.g.,* God Class) are easier to identify than others (*e.g.,* Feature Envy, or finding all occurrences of Duplicated Code) [DDN02]. Clearly it would be difficult to act on an undetected code smell.
- Removing code smells is a long-term investment that may be a hard sell to the management. Resources are scarce and there is usually a lot more to do than can ever be done.
- The benefits expected from code smell removal are not clear-cut. For example [DDVT06] results are that "the ability of participants to localize attributes[1] differed significantly among the five god class decompositions" and "moreover, this effect interacts significantly with the institution [where the student participants studied]."
- There is no consensus on how best to refactor a system to remove a given bad smell, DuBois *et al.,* again [DDVT06] note that "[they] are reluctant to accept the concept of an optimal class decomposition with respect to comprehensibility" (considering that in their experiment the institution where the participants studied had a statistically significant influence).

In such a context, we were contacted by a group at Siemens Digital Industry to help them refactor a legacy system that presented some well known code smells. Two main concerns were with the many circular dependencies that existed between the C++ classes of the system and the presence of God classes (over ten thousand lines of code, hundreds of methods and/or attributes). As is often the case, both problems are related because a very large class will tend to have many dependencies and many clients, which increase the probability that it is in circular dependency (as well as its dependencies and clients).

In collaboration with one software architect, we designed a software visualization (Internal Usage Map) to break God Classes based on colouring methods according to the attributes they access. Through experience over three years, the software

[1] Their measure of class comprehensibility.

architect formalized a process that allows her to propose a decomposition of a god class in one or two hours even without prior knowledge of the class (she does have a general knowledge of the system and the application domain).

In this paper, we detail the God Class Decomposition process (Section III) and present the visualization tool (the Internal Usage Map, Section IV). We exemplify this process and tool on several real cases by reporting and discussing their results (Section V). However, for confidentiality reasons, source code entities were anonymized and source code or pictures are blurred.

II. God Class smell

The phrase *Code Smell* was introduced by Martin Fowler and Kent Beck [FBB+99] in the book "Refactoring: Improving The Design Of Existing Code." It embodies the idea that experienced developers have a "feel" for good or bad code design and the later (code smell) is making the code difficult to understand, change, or test.

For example, the God Class smell appears when a class is too big and is responsible for too many things [DDN02]. God classes are difficult to change because various responsibilities might be intertwined in their code. They are also difficult to test because they have too many attributes that all need to be initialized correctly, even if they are not directly related to the functionality one wishes to test.

There are two accepted types of God Class (*e.g.,* [FTSC12]): Data and Behavioral. The first "hold a lot of the system's data in terms of number of attributes," and the second "implement a great portion of the system's functionality in terms of many and frequently complex methods." We are more interested in the second type; although in our experience, both appear together (with cases over a hundred methods or attributes).

Since the introduction of the term, researchers worked on code smells to identify them, remove them, or qualify more objectively the intuition that they hinder comprehension and evolution. Two decades after their introduction, they are still an active domain of research (*e.g.,* [Sha18]) and some of the smells themselves became a research topic of their own (*e.g.,* Clone Detection, [DRD99], [SK16]).

In this paper we are more particularly concerned with the God Class code smell and its treatment in a real industrial system. We will first briefly review some research in code smell detection and removal before considering their treatment in industry.

A. Detecting Code Smells

From the very beginning (*e.g.,* [DRD99]) up to recently (*e.g.,* [Sha18]) an important issue in code smell research was to identify these smells as accurately and completely as possible [DDN02], [LM06], [SSS14].

In his survey [AD15], Dallal reports six identification techniques. The paper focuses on the slightly different notion of refactoring opportunities. Refactoring opportunities and code smells are close in their inconveniences and treatment. The former might be considered as a superset of the later as it does not always stem from quality issues but could also be linked, for example, to future opportunities.

To Dallal's six techniques, we add a seventh reported by Fokaefs *et al.,* [FTSC12]:

Metrics: Using software quality metrics such as cohesion, size, similarity, or complexity to identify some of the code smells. This is the most common identification technique. It should be noted that some work characterizes a class as a God Class based on size metrics. God classes are not only classes with many attributes and methods but they define the behavior and exhibit complex control-flow of a large part of an application.

Pre-conditions: Other approaches use pre-conditions, such as "does not use local variables," or "rarely invoked," appearing jointly on the same class or methods to detect refactoring opportunities (*e.g.,* Extract Method).

Cluster: Clustering methods based on their measured similarity is also an option;

Graph: Dependency graphs can be used to identify extract interface refactoring opportunities;

Slicing: Slicing the program dependence graph of a method may help identifying extract method refactoring opportunities;

Dynamic Analysis: Analyzing method traces can be used to identify move method refactoring opportunities.

Visualization: One can visually present software entities in such a way that code smells will appear more clearly. For example showing all classes of a system with their interdependencies could help identifying Circular Dependencies problems.

Fokaefs *et al.,* [FTSC12] specifically list several papers that identify God Classes: Metrics based [TK03], [TM05]; Pre-conditions based [DDN02]; dependency Graph based [CXS04], [JJ09], [KVGS09]; Clustering based [FTCS09]; Visualization based [SSL01]. All these papers boil down to the idea that God Classes are big (many methods, attributes, and lines of code), have many dependencies, or their method-to-attribute-access graph is not connected.

B. Refactoring God Class Smell

Once a code smell is discovered it is only natural that one would consider removing it. One refactors a God Class by [DDV04] "[separating] the responsibilities, [extracting] those groups of methods and attributes that neither use nor are used by other methods or attributes." But it is rarely the case that responsibilities in a God Class are clear-cut. Typically one method may implement part of several responsibilities and access attributes that should be separate in several new classes.

Even if "separate responsibilities" was not in itself a subjective notion, one such responsibility might also be considered too small (few attributes and methods) to deserve a class of its own. We had the case at Siemens where one small reponsibility was deemed "not worth the effort" and kept together with another larger one (see Section V).

We came to the conclusion that god class refactoring should be done in two steps that are rarely explicitly recognized in

literature: (1) planning the changes, and (2) actually performing them. The same conclusion was reached by Malavota *et al.,* [MLM$^+$13]. This seems natural in their case (architectural refactoring), but is much less acknowledged in our. In our case, the planning step is important as, for example, there might be several possibilities to break down a God Class or to combine several Code Clones into one (or more) generic function(s).

Other papers propose something approaching a planning phase by prioritizing refactorings (*e.g.,* [FTCS09], [FTSC12], [MCK15]). This solution, unfortunately, does not easily allow one to prepare the removal of a given code smell as it could require several individual refactorings, each one with different priorities and mixed with other unrelated actions.

For example, Fokaefs *et al.,* [FTCS09] propose to decompose God Classes using a clustering algorithm. Their stated goal is the identification of Extract Class opportunities, which could be seen as a planning phase. But this "plan" consists in extracting one class from a God Class which is very different from planning the breaking down of a God Class in multiple new classes.

DuBois *et al.,* [DDV04] note that "Most of the time, the resolution of a refactoring opportunity leads to the advent of new opportunities. [...] the refactoring process is a dynamic process which requires continuous re-evaluation." Thus performing one refactoring at a time might engage the code base on a path leading to sub-optimal removal of the initially targeted code smell.

In the next section, we discuss removing code smells (and particularly God Class) in an industrial setting. From the conclusions already drawn and specific constraints linked to the setting, we list requirements for a God Class removal tool.

C. Code Smells in Industry

In our experience (*e.g.,* [DDN02], [BCC$^+$15], [MMBD$^+$09][2]), code smells are found in many industrial systems. Developers are often aware of the quality issues their systems exhibit but depending on the particular code smells, they might or might not be aware of all instances of these smells (*e.g.,* Code Duplication, Dead Code).

God Classes, that we are considering in this paper, are easy to spot. Developers that understand this antipattern know how to identify it and where occurrences can be found in their code. Therefore, we do not believe any special means is required to identify God Classes.

As suggested in the previous section, removing a God Class smell is much more complex than what the literature usually acknowledges. Such large code restructurings are not just a matter of applying several Extract Method, Extract Class, or Move Method refactorings. Each of these refactorings implies altering dependencies, within the God Class itself and for its clients, in ways that must be monitored to avoid unexpected consequences. A naive planning strategy could be achieved using some revision control system. One would actually do the changes in the code and revert back to a previous situation if one discovered the obtained design is not satisfactory. But this would be extremely costly, to the point of being impracticable. In our experiments, one God Class refactoring that took less than two hours to plan, took a week to implement.

[2]references anonymized for double blind review

The best approach seems to allow planning the new design and then perform the changes to reach the planned goal. In the case of God Class, planning means:

- Identifying up front the different responsibilities to extract;
- Assigning methods and attributes of the God Class to these responsibilities;
- Possibly decomposing existing methods into the different responsibilities they take part in;
- Understanding how the new classes will interact together to replicate all the behaviour of the God Class, and;
- Understanding how they will interact with the rest of the system, and particularly how they will be used by the clients of the God Class.

The Orion planning tool [LDDF11] would be a possible answer to the last two needs.

Only when a satisfying design is reached would someone consider applying refactorings to the God Class to implement this design in the source code. Ideally, an intelligent tool would be able to automatically deduce and execute most of the refactorings from the final design obtained and the path to get to it. Yet some manual fine tuning seems inevitable, for example when it comes to decomposing an existing method in the God Class into several methods for the refactored classes.

In the following section we present the God Class Decomposition process that was formalized by Siemens software architect through practice.

III. God Class Decomposition Process

The aim of the God Class decomposition process is to find, within the God Class, smaller, more cohesive, sets of methods and attributes that could be made into new classes. The intuition is that each subset of methods will collectively access a subset of the God Class attributes.

It is important to stress at this point that the proposed process is not an algorithm in the sense that it cannot be fully automated. It relies on the appreciation of the developer in several steps. This can be based on some hints as naming conventions or regularities in the source code indicating similar organisation.

We strongly believe that fully automated solutions cannot solve many complex problems in software engineering that require an extended knowledge of the application domain, the organization in which the software runs, its history, the various stakeholders including the developers, etc. All these aspects influence the way software engineers develop software [KPP$^+$02], [AdOdSBD07] and should be taken in consideration also when refactoring a code smell.

We rather favor an approach where the tools form a "software exoskeleton" helping the developers in their work but leaving them in full control of the task. In a sense, modern

IDEs and their refactoring engines are a good image of what we are aiming for, but such new extensions should be created.

At a high level, the God Class decomposition process is composed of four steps. Two of them are complex subprocesses described in the following subsections. In our description we call *responsibilities* the future classes to be isolated and extracted from the God Class.

Cleaning: This step aims to reject from the God Class attributes and methods that have little or no interest.

First, there is, the obvious case of dead methods that are not called anywhere in the program. These methods are removed from the God Class and will not be migrated to the new classes.

Similarly, "dead attributes," not used in the class (or outside it) are removed. An attribute may become dead after a dead method using it has been removed.

Attributes that are only used in one method can easily be converted to local variables. This allows reducing the amount of entities to manage at the class level.

Finally attributes that are accessed in too many methods (rule of thumb: more than a third of the God Class' methods) are not considered either. Because they are used a lot, there are few chances that they can be extracted into one of the responsibilities. On the contrary, it must be expected that they would create links between these responsibilities and cluster them together. These attributes are not considered in the process; in a later phase, they are gathered in a "Common class." This is a special new class that will often be referenced by all the new responsibility classes. In a sense, it is the core part of the God Class that cannot be assigned to any individual responsibility.

Marking attributes: This step is the main part of the process. It is repeated to identify each responsibility (that will be externalized as a new class) by assigning attributes to it. Because we use a visual tool that marks such responsibilities with separate colours (see Section IV), we also call them "colors." For example in the cleaning step, attributes having too many accessors are marked in yellow (color chosen for no special reason).

The color helps to visually mark the attributes/methods belonging to a responsibility while one may not yet fully understand the essence of this responsibility: what it represents in terms of application domain concept (or programming concept).

This step will be detailed in Section III-A.

Marking methods: Once all attributes are marked, or the architects are satisfied that all responsibilities of interest have been completely identified (*i.e.,* all their attributes were marked), they mark methods with the color of their responsibility.

We detail this step in Section III-B.

Finalizing: The methods and attributes remaining at the end are the conflicting ones (see sections III-A and III-B). They are gathered in the special Common class that represents the core of the God Class.

In one occasion, only one responsibility was identified (*i.e.,* the God Class was not entirely broken down). In such a case, the remaining attributes and methods were those that did not pertain to the responsibility and the Common class was actually the God Class itself, deprived of this responsibility.

A. *Marking Attributes Subprocess*

This subprocess aims at marking all attributes that participate in a given responsibility. Visually, the responsibility is marked by an identifying color (Section IV). Abstractly, the subprocess consists in marking attributes through the methods that access them. This helps to ensure that the responsibility (*i.e.,* its methods and attributes) is coherent. One chooses some method accessing unmarked attributes and tries to mark these attributes with the current responsibility. When all attributes accessed by a method are marked, one chooses another method with unmarked attributes. This subprocess loops as long as the software architect believes there are other attributes to be assigned to the current responsibility.

Ideally, one would execute this subprocess for each responsibility in turn, each time, with a different color to mark the current responsiblity. But, like the entire God Class decomposition process, this subprocess is iterative and incremental. The software architect marks attributes one at a time while continuously checking how this reflects on all methods. Building an understanding of the God Class, the software architect tries to ensure the coherence of each responsibility (*i.e.,* color) as it is emerging. Obviously, things are not clear-cut and one may have to come back to a previous responsibility when dealing with another one.

The steps of this subprocess are as follows:

Initializing: One chooses a method accessing some attributes not yet marked. What method to choose depends on whether this is the inception of the responsibility identification or not.

If this is the start of the responsibility extraction, it is best if the method accesses only unmarked attributes.

If the current responsibility already has some attributes assigned to it (marked with its color), one chooses a method with both unmarked and marked attributes. It is best if this method does not access attributes of another responsibility.

In both cases (responsibility inception or not) it is best to choose a method accessing enough, but not too many attributes (rule of thumb: 4 to 8 attributes accessed), and for which the attributes are not too often accessed (rule of thumb: less than 10 methods in total accessing them). The rationals are that if the method accesses too many attributes or if an attribute is accessed by too many methods, then chances are higher that it takes part in several responsibilities. So one wants to leave it for future treatment.

If the method accesses too few attributes, it will will be hard to progress because it only deals with a limited part of the responsibility. Consider for example choosing a

getter/setter method, it would access only one attribute and as such would not be very helpful in gathering all attributes pertaining to the same responsibility.

Looping over attributes: Mark all (unmarked) attributes accessed by the chosen method.

If one of the attributes accessed by this method was already marked with another color (*i.e.,* accessed by another method considered in a previous application of the subprocess), one should decide whether this attribute stays in the previous responsibility, migrates to the new (current) responsibility, or is conflicting because it belongs to both responsibilities. Of course, this can be decided by considering the meaning of the responsibilities and the particular attribute. But, especially at the beginning of the whole process, the meaning of the current color might not be clear yet, or the meaning of the attribute might not be known.

In such a case, the decision can be facilitated by comparing how many accesses to the attribute each method makes. One can also look at the methods accessing this attribute and what are their dominant colors (the dominant color of the attributes they access). The goal is to have the least possible methods in conflict.

If the current method and other methods make a comparable number of accesses to the attribute, then this one is said to be conflicting and marked as such.

If another method makes much more accesses than the current one, then the attribute remains of the same (previous) color. As a consequence the current method is in conflict (treated in Section III-B).

If the current method makes many more accesses than the other ones, then the attribute should probably be remarked with the new color. But before that, one must check whether changing the color of the attribute would not create conflicts in too many of the other methods. If this would be the case, again the attribute remains marked with the same previous color, and the current method has a conflict.

Finally if the current method makes many more accesses than the other ones, and these other ones would not be conflicting too much from a change of color of the attribute, then make the change.

B. Marking Methods Subprocess

This step starts when all attributes of the responsibilities of interest are marked. When all attributes accessed by a method are of the same color, this method itself is marked with this color. Special considerations must be given to methods accessing attributes marked with several different colors (including the special "in conflict" color) and methods accessing no attributes.

Methods in conflict: Methods accessing attributes marked with several colors (in conflict methods) need to be broken down. This can be done by looking at the source code of the method and identifying the individual statements that make these accesses. The goal is to create new methods that will access only attributes marked with one color (or with the special "in conflict" color). Here the software architect will need to analyze the code to build working methods according to the colors of the attributes. In our experience, this is not overly difficult.

Methods accessing no attribute: For these methods one looks at their incoming invocations and follows a subprocess similar to marking attributes (Section III-A). If the invokers of a method are all marked in the same color, the method takes this color. If the invokers of a method have different colors, it is marked as "in conflict" and must be decomposed (previous step).

IV. Tool Description

All the steps of the process described above are helped by a tool that we implemented. This tool is based on the Moose platform (*e.g.,* [DLT00], [BAD12]) which already includes features such as source-code modelling, query engines on the model, or agile data visualization. Therefore, Moose is used to create an internal model of the source code: a graph of classes, methods, variables, etc. and how they interact (classes define methods, methods access variables and invoke other methods, . . .).

As noted in the previous section, the tool strongly relies on visualizations as a mean to materialize the responsibilities to extract. We used for this the visualization the scripting engine of Moose[3]. Fokaef [FTSC12] argues that "The fundamental shortcoming of visualization based approaches is that [...] they do not scale up." We disagree with this opinion and past research proved that visualizations may handle large amount of data efficiently and help users to more easily make sense out of large amount of data (*e.g.,* [DGK06], [Lan03]).

We present the tool through several screenshots that illustrate:

- Marking of attributes and methods (Section IV-A);
- Decomposing methods in conflict (Section IV-B);
- Reviewing the final design of the future classes and how they interact between themselves and with the rest of the system (Section IV-C)

A. Marking Attributes and Methods

Marking attributes and methods is done through the Internal Usage Map which displays the methods of a God Class, with the following conventions (see also Figure 1):

- The figure shows all the methods of a class and the attributes they access;
- Outer squares represent methods (16 in the figure);
- Inner squares represent the attributes accessed by each methods;
- One attribute may appear in several methods, like the green attribute with gold border appearing in six methods (number 1, 2, 4, 5, 6, and 10);
- The width of an attribute is an indication of the strength of the access to this attribute by the method. If the method

[3] agilevisualization.com

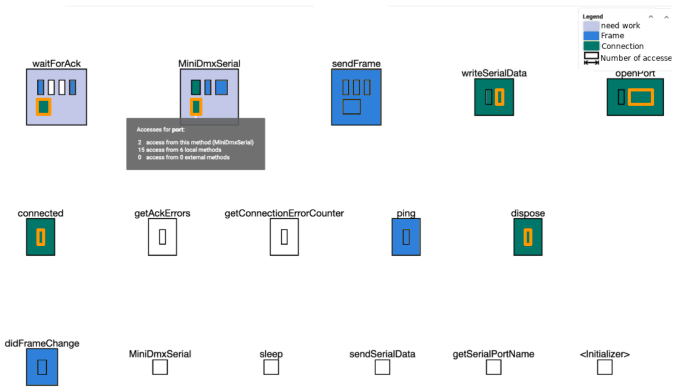

Fig. 1. A simple example of the Internal Usage Map for a small class: Outer squares are methods, inner squares are accessed attributes. The same attribute can appear as an inner box in several methods when they all access it. The inner boxes with gold border actually represent all the same attribute. Two responsibilities (green and blue) were identified by the software engineer. Light grey means "need work", the second and third methods have no identified resposibilities yet and they access attributes with different identified responsibilities.

makes a lot of use of the attribute, it will be represented by a wider inner square. Note, therefore that an attribute can be thin in a method and wide in another one;

- Clicking on one attribute (*e.g.,* last attribute in the second method) highlights it (gold border) in all the methods that access it. It also gives detailed information on these accesses. In the Figure, the dark grey box tells us that the method *MiniDmxSerial* (the second method) makes two accesses to the attribute *port*, which receives 15 accesses in total from six methods, and no access from outside the class.

An important concept of Moose that we used is the "Tag." [GAE$^+$17] Tags are logical concepts, attached to source code entities. They are first class entities of the model just like actual classes and methods. Tags materialize the comprehension of the system that software engineers have. Each tag has a label (or rational) and a color to identify it visually.

For example, in Figure 1, the notion of "Frame" (blue) and "Connection" (green) are two application domain concepts attached to attributes and methods of the class. Tags may also model other things than application domain concepts. In the figure, the tag "need work" (grey) models the fact that the first two methods need to be split because they access attributes linked to "Frame" and "Connection" (they are in conflict).

White attributes and methods have not been marked yet. The next step (Marking attributes in Section III), in the example of Figure 1, would probably be to mark one of the two accessed white attributes in the fifth and sixth methods. It might actually be the same attribute and marking it in one of the method would also color it in the other method. One would then mark these two methods (step Marking methods in Section III) with the same color because they only access an attribute of this color.

The last five methods do not access any attribute. They would need to be studied individually to understand where they fit (step Methods not accessing any attribute in Section III-B).

In Figure 2, we illustrate a real case of marking attributes and methods of a God Class. There are 244 methods and 98 attributes in this class. Three methods are circled in red to illustrate the following points:

- The last one is the simplest; it accesses only brown attributes and was therefore also marked in brown. This is an example of a clearly identified responsibility (at least for this method, other brown methods are not so clear);
- The first circled method accesses attributes of many different colors. It is in conflict and marked as such. Although it is not a strong requirement of the process, the software architect often uses two colors for the methods in conflict: yellow methods go in the Common class (just like yellow attributes); orange methods are those that should be decomposed. Here the method is an initialization one and it is marked in orange to indicate that each attribute initialization should go in the respective future classes of the attributes.
- The middle circled method was marked in blue because it accesses many blue attributes. However one can spot three yellow attributes and one green one. The yellow attributes are already in conflict and as such they have less impact on the choice of a color for this method. The green attribute means this methods accesses an attribute from another responsibility. It will therefore need to be changed to remove this access (the thin attribute indicates that there are not many accesses to this attribute in this method). One can either split the method in two, or use an accessor method in the future green class to access the attribute.

B. Decomposing methods

Figure 3 shows the source code visualization of a conflicting method (on a Java example for confidentiality reasons). In this detailed visualization of the code, accesses to marked attributes appear with the color of the attribute. This greatly helps deciding whether the method should be split, and how to do it.

C. Reviewing the Final Design

By materializing the different responsibilities with tags that are first-order entities in the Moose model, the software architect is able to visualize the impact of the refactoring before actually doing it. This helps checking that no unforeseen dependency will arise from the modifications.

Figure I shows the interactions of the future classes (from their tags) between themselves and with the currently existing classes (other than the God Class itself).

The three larger squares (pink and purple) are the future classes. They are unfolded to show the dependencies from/to each of their methods. The smaller yellow boxes are the currently existing classes. Dependencies between methods (and therefore between classes) are extracted from the current

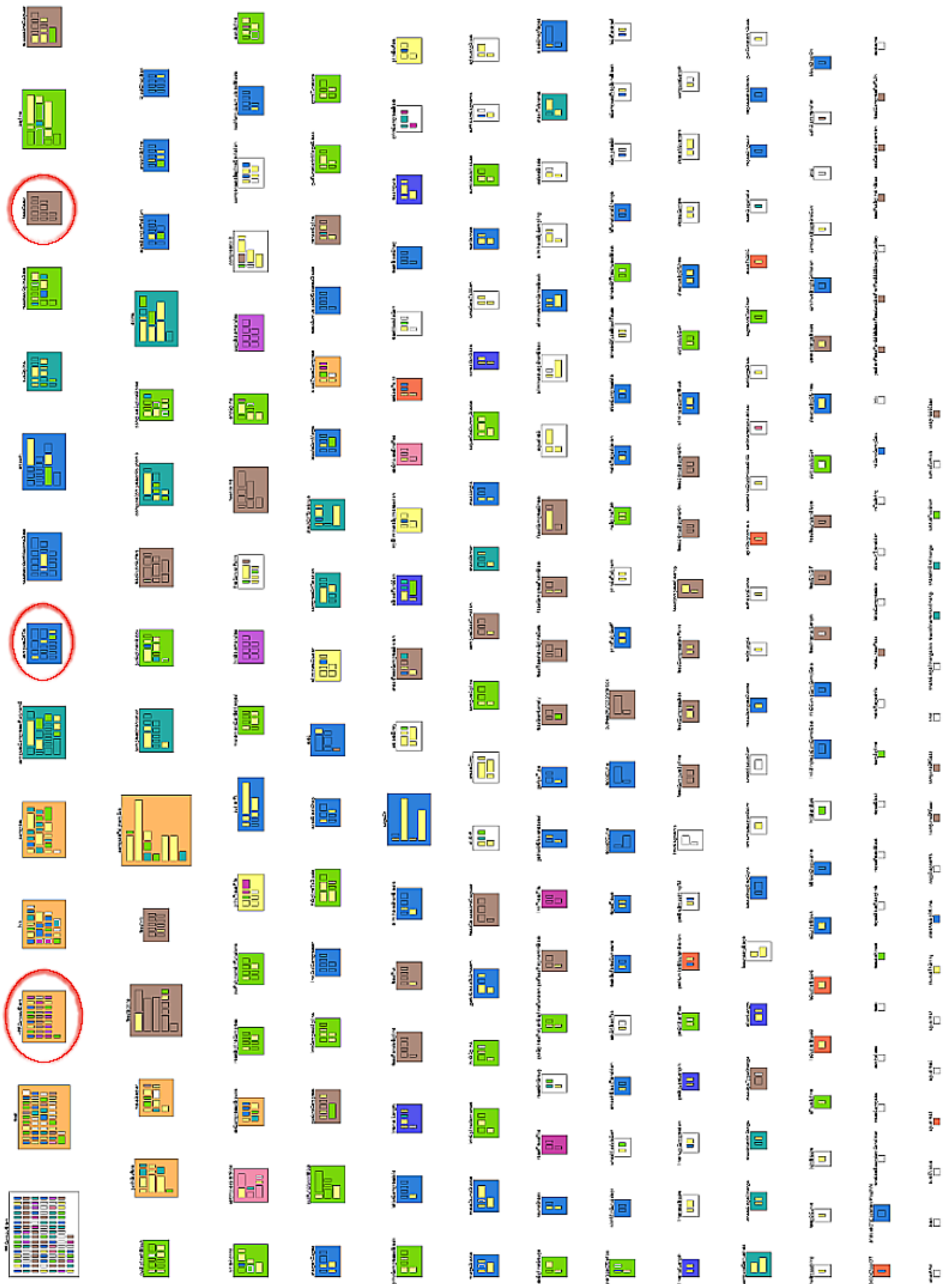

Fig. 2. A real example of the Internal Usage Map for a God class: outer squares are methods, inner squares are attributes they access. Colors mark responsibilities identified by the software engineer (visualization rotated, top on left)

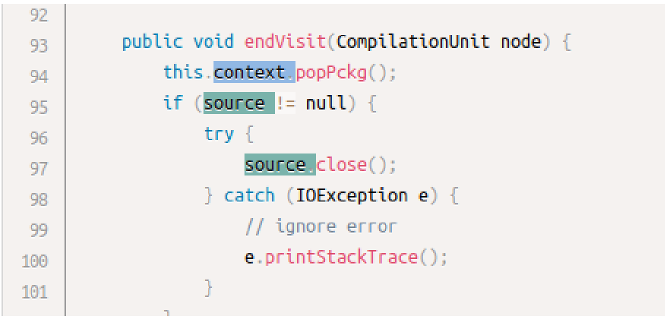

Fig. 3. Visualization of the code of a method in conflict: Accesses to marked attributes appear in the color of the attribute

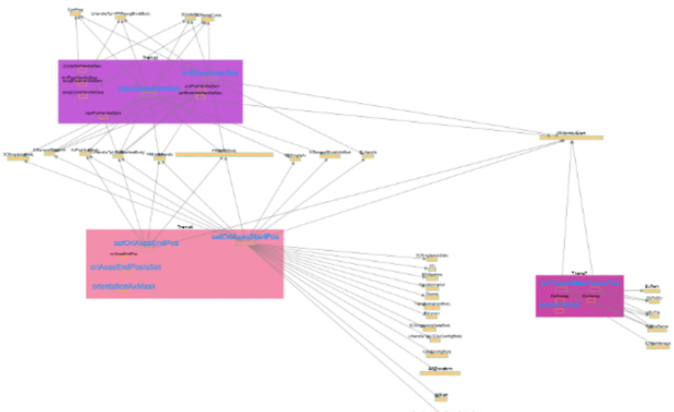

Fig. 4. Visualizing the interactions between future and current classes (names blurred for confidentiality)

source code assuming it will not change. This assumption is never entirely true, but the visualization is nevertheless of great help to understand the impact of the planned refactorings.

V. Case studies

During the three years of collaboration between Siemens Digital Industry and our research team, several God Class decompositions were performed either on the source code, or as training activities to explain the process and the tools to the software engineers.

We will now give some descriptive data on the software system on which the development team works, and on the God Classes that were refactored or for which a refactoring was "planned" (training sessions).

A. The Software System

The software system is the heart of several industrial drilling machines. It includes a parser for a language used to describe the path the drill must follow, as well as an interpreter of the language to execute the instructions. The speed of movements, and particularly direction changes, has to be controlled to prevent breaking the cutting head.

The system is 30+ years old and written in C++. It has about 10,000 classes and over 2.5 MLOC, and 224 classes

	min.	max.	average
9 531 Classes			
LOC	14	59 981	1 039.4
#methods	0	1 252	8.2
#attributes	0	456	4.2
77 880 Methods			
LOC	0	15 238	30.4

TABLE I
DESCRIPTIVE DATA ON THE C++ SOFTWARE SYSTEM CONSIDERED

are considered God Classes (over 50 methods). Table II gives some descriptive data on the system.

The development team includes 40 engineers scattered over three locations in Germany. Many of these developers are physicists (as opposed to people specifically trained in computer science).

The reasons the developers want to refactor the God Classes include:

- When there is a bug to correct, developers are reluctant to deal with a God Class because the correction could easily create a new bug;
- Understanding a God Class takes more time;
- The system implements sophisticated algorithms that deal with complex notions of physics and kinematics. It is very difficult to replace one algorithm by another because the said algorithm is scattered over too many methods and intertwined with other processing;
- It is costly to test these classes.

B. Real Refactorings Results

We give here results for two real examples of God Class refactorings that were performed at Siemens by a software architect.

The first one is God Class "**Real_1**" (fictitious name). We give in Table III some data on the refactoring. As already mentioned, it took less than two person-hours to plan (identification of the responsibilities) but one person-week of work to actually implement. This week of work includes time to create some additional tests, migrate existing and added tests and make them all pass.

We mentioned in Section II-C that a tool was needed to automatically deduce and execute the refactorings required to reach the design obtained after the planning phase. The time to implement this refactoring highlights the urgent need for such a tool and this is one of the research paths we wish to explore. It would also be nice to be able to generate automatically some tests to check the validity of the refactoring. This could be doable if restricted to one particular God Class as used by its current clients.

We give some data on this refactoring in Table III and show a UML diagram of the resulting design in Figure 4.

When implementing the refactoring, NewClass1 and NewClass3 were kept together. The rational was that New class 1 was two small (4 attributes, 2 methods) and not worth the effort of extracting it. One could consider that the merged NewClass1+3 is not much smaller than the initial God Class

	Attributes	Methods
Real_1	47	58
CommonClass	4	13
Supervisor class	5	10
NewClass1	4	2
NewClass2	16	16
NewClass3	18	49

TABLE II
NUMBER OF ATTRIBUTES AND METHODS OF GOD CLASS "REAL_1" THAT WAS REFACTORED (TOP) AND OF THE RESULTING FIVE CLASSES AFTER REFACTORING.

	Attributes	Methods
Real_2	146	169
Common class	109	134
NewClass1	6	14
NewClass2	25	47
NewClass3	2	5
NewClass4	11	26

TABLE III
NUMBER OF ATTRIBUTES AND METHODS OF GOD CLASS "REAL_2" THAT WAS REFACTORED (TOP) AND OF THE RESULTING FIVE CLASSES AFTER REFACTORING.

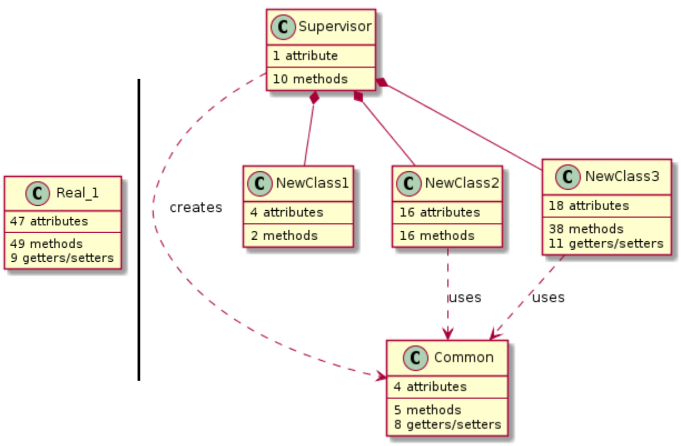

Fig. 5. UML diagram before (left) and after (right) of the refactored God Class "Real_1"

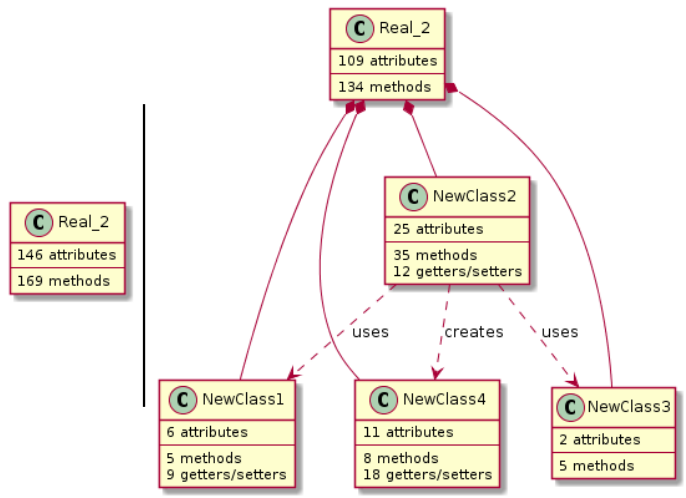

Fig. 6. UML diagram before (left) and after (right) of the refactored God Class "Real_2"

(49+2 methods instead of 58), but it does have less than half the number of attributes (18+4 instead of 47).

Note that there are many more methods after the refactoring (90) than before (58). This is expected as (i) some methods were split because they were in conflict; (ii) new accessor methods might be required to allow one new class to access an attribute assigned to another new class.

Similarly, the number of attributes after refactorings might not be the same as before since (i) some attributes may be converted to local variables; (ii) attributes will be created in the new classes to hold references to instances of the other new classes, one instance of the God Class will be replaced by instances of several classes that need to collaborate.

Another application of the process (God Class "**Real_2**") was performed in two phases. First one algorithm from the class was extracted with the idea that it should be replaced later by a more efficient one. Some time latter, the God Class was further decomposed. The result presented here is not considered final yet and the Common class is still a God Class.

We give some data on this refactoring in Table IV and show the final UML diagram in Figure 5. The algorithm first extracted from God Class "Real_2" is the NewClass2.

C. Training Refactorings Results

Some of the refactorings were performed during training sessions to show the tool to members of the development group. The training sessions last about 60 to 90 minutes and include an initial presentation giving a high level view of the process and some explanation of the tool functionalities (including other functionalities than the God Class decomposition).

The main part of a training session consists of a practical exercice with a real God Class that the trainee group as a whole decomposed. The work remains at the planning stage and the decomposition is not actually put in practice.

God Class "**Training_1**" (described in Table V) has 110 attributes and 75 methods. After the refactoring, seven new classes were proposed one of which is a Common class. The refactor planning is described in Table V. All attributes were redistributed among the seven new classes. Seventeen methods are in conflict in the final design and would required a detailed analysis which was not done during the training session.

God Class "**Training_2**" (described in Table VI) has 107 attributes and 82 methods. The refactoring plan produced six new classes including the Common class. One observes that the planned Common class is still rather big and could benefit from a more detailed analysis.

A final God Class "**Training_3**" is described in Table VII.

VI. RELATED WORK

The topic of Code Smell spawned innumerable publications. As explained in Section II-A many concern the identification of Code Smells in the source code. Several papers were already

	Attributes	Methods
Training_1	110	75
Cleaning step	*6*	*5*
CommonClass	10	3
SupervisorClass	14	4
NewClass1	9	1
NewClass2	16	7
NewClass3	32	39
NewClass4	7	1
NewClass5	16	15

TABLE IV
NUMBER OF ATTRIBUTES AND METHODS OF GOD CLASS "TRAINING_1" AND THE RESULTING PLANNED NEW CLASSES

	Attributes	Methods
Training_2	107	82
CommonClass	56	47
NewClass1	13	15
NewClass2	9	2
NewClass3	4	3
NewClass4	16	12
NewClass5	9	3

TABLE V
NUMBER OF ATTRIBUTES AND METHODS OF GOD CLASS "TRAINING_2" AND THE RESULTING PLANNED NEW CLASSES

cited and we will not come back to them. Our position is that, at least for the case of God Class, such techniques are not needed as developers are often aware of the presence of the smell or can easily find them back.

Similarly, for code smell removal there are many publications. We noted that many fail to recognize that this should be done in two steps: planning the refactoring and implementing it. Most publications consider the removal of code smells under the aspect of applying a refactoring (such as Extract method) and choosing where to apply it (*i.e.,* identification of the smell). For some large smells like God Class, requiring several refactorings, an elaborated plan should be made before starting the refactoring.

Another mistake commonly found in literature is to aim for full automation [**?**], [**?**]. We believe good design is a too fuzzy notion, relying on many factors external to the code to be completely automated. In this line of research, we may cite Fokaefs *et al.,* [FTCS09], [FTSC12] who propose

	Attributes	Methods
Training_3	60	74
CommonClass	10	20
NewClass1	15	7
NewClass2	12	15
NewClass3	6	21
NewClass4	6	6
NewClass5	2	2
NewClass6	9	3

TABLE VI
NUMBER OF ATTRIBUTES AND METHODS OF GOD CLASS "TRAINING_3" AND THE RESULTING PLANNED NEW CLASSES

to decompose God Classes using a clustering algorithm. We already commented (Section II-B) that the solution is geared toward generally improving the cohesiveness of classes one refactoring at a time rather than breaking down a given God Class.

Joshi and Joshi [JJ09] propose using a lattice of concepts to identify cohesive classes. The issue with this solution is that for real God Classes (we have examples with over a hundred methods and attributes) the lattice would become so big, it would be impractical to analyze.

DeLucia *et al.,* [LOV08] propose considering the names of attributes and methods to identify cohesive concepts, but this would imply some consistent naming scheme has been used, which is rarely the case.

VII. CONCLUSION

Code smells are a recognized nuisance when it comes to evolving software systems. We were contacted by a group at Siemens Digital Industry Division to help them remove God Class code smells from a big software system. Some classes have over one hundred methods and attributes and thousands of lines of code.

We found that the published literature did not allow us to address their situation, namely:

- Many articles focus on detecting code smells which is not necessary for the case of God Classes. They are usually very prominent and easy to spot.
- Solutions often fail to acknowledge that there are two phases: first planning the code smell removal, then doing it. An independent planning phase is important to make sure one will be able to reach a satisfactory result before commiting to any change. Applying the code refactorings might be a long process (a person-week of work in our case for two hours of planning) that should not be undertaken without knowing where one goes.

We describe an iterative God Class decomposition process based on a visualization (Internal Usage Map) of the methods of a class and the attributes they access. By marking attributes and methods that access them with colors, the software architect can identify independent responsibilities. A tool based on the Moose software analysis platform helps applying this process and visualizing the result of the planned changes before they are actually applied.

This process and this tool were applied on several classes at Siemens and we described some of these experiments. The results on this industrial setting are good. We see no reason why it should not be the same in other settings (*e.g.,* open source software) as no particular characteristic of the company or system seem to be at work here. Of course this should be submitted to the test of reality and experiment should be performed in this direction.

Other possible future work we are considering is how to decompose conflicting methods that access attributes pertaining to different identified responsabilities. In the specific case of Siemens software, we noted somemethods that implement several strategies controled by testing (if-then-elseif) the value

of a single variable. We hope to be able to help detect these cases, split the startegies in different methods and convert the initial method into a dispatcher.

References

[1] M. Abbes, F. Khomh, Y. Gueheneuc, and G. Antoniol, "An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension," in *2011 15th European Conference on Software Maintenance and Reengineering*, Mar. 2011, pp. 181–190.

[2] B. DuBois, S. Demeyer, J. Verelst, and M. Temmerman, "Does God Class Decomposition Affect Comprehensibility?" in *Proceedings of the IASTED International Conference on Software Engineering*. Innsbruck, Austria: IASTED/ACTA Press, 2006. [Online]. Available: https://www.researchgate.net/publication/220901508_Does_God_Class_Decomposition_Affect_Comprehensibility

[3] S. M. Olbrich, D. S. Cruzes, and D. I. K. Sjoberg, "Are all code smells harmful? a study of god classes and brain classes in the evolution of three open source systems," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM'10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2010.5609564

[4] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Information and Software Technology*, vol. 58, pp. 231–249, Feb. 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584914001918

[5] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, Oct. 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121212001057

[6] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006. [Online]. Available: http://www.springer.com/alert/urltracking.do?id=5907042

[7] R. Malhotra, A. Chug, and P. Khosla, "Prioritization of classes for refactoring: A step towards improvement in software quality," in *Proceedings of the Third International Symposium on Women in Computing and Informatics*, ser. WCI '15. New York, NY, USA: ACM, 2015, pp. 228–234, event-place: Kochi, India. [Online]. Available: http://doi.acm.org/10.1145/2791405.2791463

[8] J. A. M Santos, M. G. de Mendonça, C. P. dos Santos, and R. L. Novais, "The problem of conceptualization in god class detection: agreement, strategies and decision drivers," *Journal of Software Engineering Research and Development*, vol. 2, no. 1, p. 11, Sep. 2014. [Online]. Available: https://doi.org/10.1186/s40411-014-0011-9

[9] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "Methodbook: Recommending Move Method Refactorings via Relational Topic Models," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 671–694, Jul. 2014.

[10] C. Y. Chong, S. P. Lee, and T. C. Ling, "Efficient software clustering technique using an adaptive and preventive dendrogram cutting approach," *Information and Software Technology*, vol. 55, no. 11, pp. 1994–2012, Nov. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584913001481

[11] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002. [Online]. Available: http://rmod.inria.fr/archives/books/OORP.pdf

[12] A. A. Rao and K. N. Reddy, "Identifying clusters of concepts in a low cohesive class for extract class refactoring using metrics supplemented agglomerative clustering technique," *arXiv:1201.1611 [cs]*, Jan. 2012, arXiv: 1201.1611. [Online]. Available: http://arxiv.org/abs/1201.1611

[13] F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics based refactoring," in *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, Mar. 2001, pp. 30–38.

[14] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for Software Design Smells: Managing Technical Debt*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2014.

[15] R. Pérez-Castillo and M. Piattini, "Analyzing the Harmful Effect of God Class Refactoring on Power Consumption," *IEEE Software*, vol. 31, no. 3, pp. 48–54, May 2014.

[16] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[17] T. Sharma, "Detecting and managing code smells: Research and practice," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 546–547. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183460

[18] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, H. Yang and L. White, Eds. IEEE Computer Society, Sep. 1999, pp. 109–118. [Online]. Available: http://rmod.inria.fr/archives/papers/Duca99bCodeDuplication.pdf

[19] K. Solanki and S. Kumari, "Comparative study of software clone detection techniques," in *2016 Management and Innovation Technology International Conference (MITicon)*, oct 2016, pp. MIT–152–MIT–156.

[20] L. Tahvildari and K. Kontogiannis, "A metric-based approach to enhance design quality through meta-pattern transformations," in *Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.*, Mar. 2003, pp. 183–192.

[21] A. Trifu and R. Marinescu, "Diagnosing design problems in object oriented systems." in *Proceedings of 12th Working Conference on Reverse Engineering (WCRE 2005), 7-11 November 2005, Pittsburgh, PA, USA*. Los Alamitos CA: IEEE Computer Society, 2005, pp. 155–164.

[22] A. Chatzigeorgiou, S. Xanthos, and G. Stephanides, "Evaluating object-oriented designs with link analysis," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 656–665. [Online]. Available: http://dl.acm.org/citation.cfm?id=998675.999469

[23] P. Joshi and R. K. Joshi, "Concept analysis for class cohesion," in *Proceedings of the 2009 European Conference on Software Maintenance and Reengineering*, ser. CSMR '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 237–240. [Online]. Available: https://doi.org/10.1109/CSMR.2009.54

[24] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *Proceedings of the 2009 Ninth International Conference on Quality Software*, ser. QSIC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 305–314. [Online]. Available: https://doi.org/10.1109/QSIC.2009.47

[25] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing object-oriented class modules using an agglomerative clustering technique," in *2009 IEEE International Conference on Software Maintenance*, Sep. 2009, pp. 93–101.

[26] B. DuBois, S. Demeyer., and J. Verelst, "Refactoring - improving coupling and cohesion of existing code," in *11th Working Conference on Reverse Engineering*, Nov. 2004, pp. 144–151.

[27] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang, "What industry needs from architectural languages: A survey," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 869–891, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.74

[28] V. Blondeau, S. Cresson, P. Croisy, A. Etien, N. Anquetil, and S. Ducasse, "Predicting the Health of a Project? An Assessment in a Major IT Company," in *8th Seminar on Advanced Techniques & Tools for Software Evolution (SATToSE'15)*, Mons, Belgium, Jul. 2015. [Online]. Available: http://rmod.inria.fr/archives/papers/Blon15b-SATToSE-PredictingProjectHealth.pdf

[29] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues, "The squale model – a practice-based industrial quality model," in *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*, Edmonton, Canada, 2009, pp. 94–103. [Online]. Available: http://rmod.inria.fr/archives/papers/Mord09a-ICSM2009-SqualeModel.pdf

[30] J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri, "Supporting simultaneous versions for software evolution assessment," *Journal of Science of Computer Programming (SCP)*, vol. 76, no. 12, pp. 1177–1193, May 2011. [Online]. Available: http://rmod.inria.fr/archives/papers/Lava10a-Official-SCP-Orion.pdf

[31] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, "Preliminary guidelines for empirical research in software engineering," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 721–734, 2002.

[32] N. Anquetil, K. M. de Oliveira, K. D. de Sousa, and M. G. Batista Dias, "Software maintenance seen as a knowledge management issue," *Information Software Technology*, vol. 49, no. 5, pp. 515–

529, 2007. [Online]. Available: http://rmod.inria.fr/archives/papers/Anqu07a-IST-MaintenanceKnowledge.pdf

[33] S. Ducasse, M. Lanza, and S. Tichelaar, "Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems," in *Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools*, ser. CoSET '00, Jun. 2000. [Online]. Available: http://rmod.inria.fr/archives/papers/Duca00bMooseCoset.pdf

[34] M. U. Bhatti, N. Anquetil, and S. Ducasse, "An environment for dedicated software analysis tools," *ERCIM News*, vol. 88, pp. 12–13, Jan. 2012. [Online]. Available: http://ercim-news.ercim.eu/images/stories/EN88/EN88-web.pdf

[35] S. Ducasse, T. Gîrba, and A. Kuhn, "Distribution map," in *Proceedings of 22nd IEEE International Conference on Software Maintenance*, ser. ICSM'06. Los Alamitos CA: IEEE Computer Society, 2006, pp. 203–212. [Online]. Available: http://rmod.inria.fr/archives/papers/Duca06cDistributionMap.pdf

[36] M. Lanza, "CodeCrawler — lessons learned in building a software visualization tool," in *Proceedings of CSMR 2003*. IEEE Press, 2003, pp. 409–418. [Online]. Available: http://scg.unibe.ch/archive/papers/Lanz03aLessonsLearned.pdf

[37] B. Govin, N. Anquetil, A. Etien, S. Ducasse, and A. Monegier Du Sorbier, "Managing an Industrial Software Rearchitecting Project With Source Code Labelling," in *Complex Systems Design & Management conference (CSD&M)*, Paris, France, Dec. 2017. [Online]. Available: https://hal.inria.fr/hal-02095200

[38] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *34th International Conference on Software Engineering (ICSE '12)*.

[39] E. Murphy-Hill, C. Parnin, and A. P. Black, ""ow we refactor, and how we know it," *IEEE Transactions on Software Engineering*, 2012.

[40] A. D. Lucia, R. Oliveto, and L. Vorraro, "Using structural and semantic metrics to improve class cohesion," in *2008 IEEE International Conference on Software Maintenance*, Sep. 2008, pp. 27–36.